EBERHARD KARLS
UNIVERSITÄT
TÜBINGEN

FACULTY OF
HUMANITIES
Department of General and Computational Linguistics

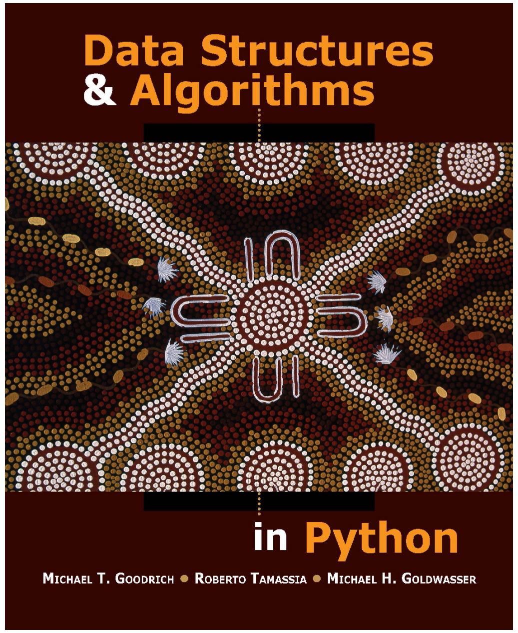
Graphs

Data Structures and Algorithms for CL III, WS 2019-2020

Corina Dima
corina.dima@uni-tuebingen.de

Data Structures & Algorithms in Python

MICHAEL GOODRICH
ROBERTO TAMASSIA
MICHAEL GOLDWASSER



14.1 Graphs

- The Graph ADT

14.2 Data Structures for Graphs

- Edge List Structure
- Adjacency List Structure
- Adjacency Map Structure
- Adjacency Matrix Structure

Co-authorship Graph – undirected graph

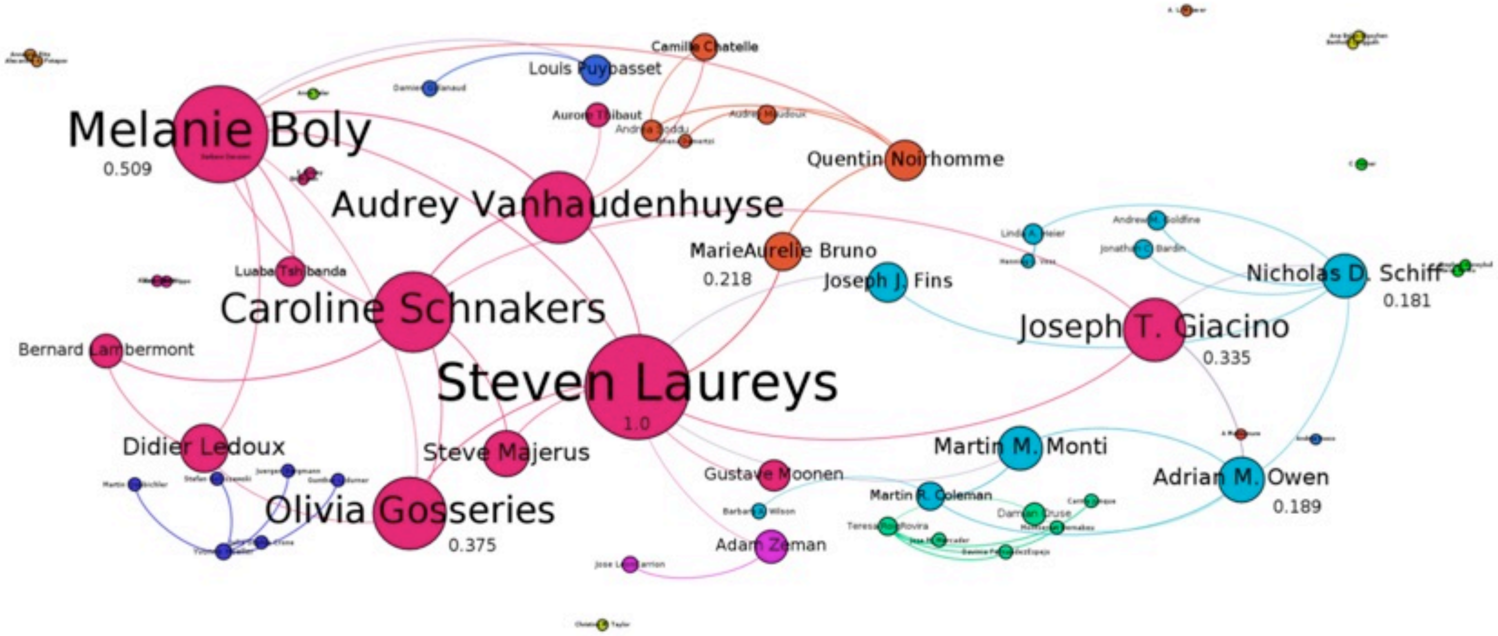


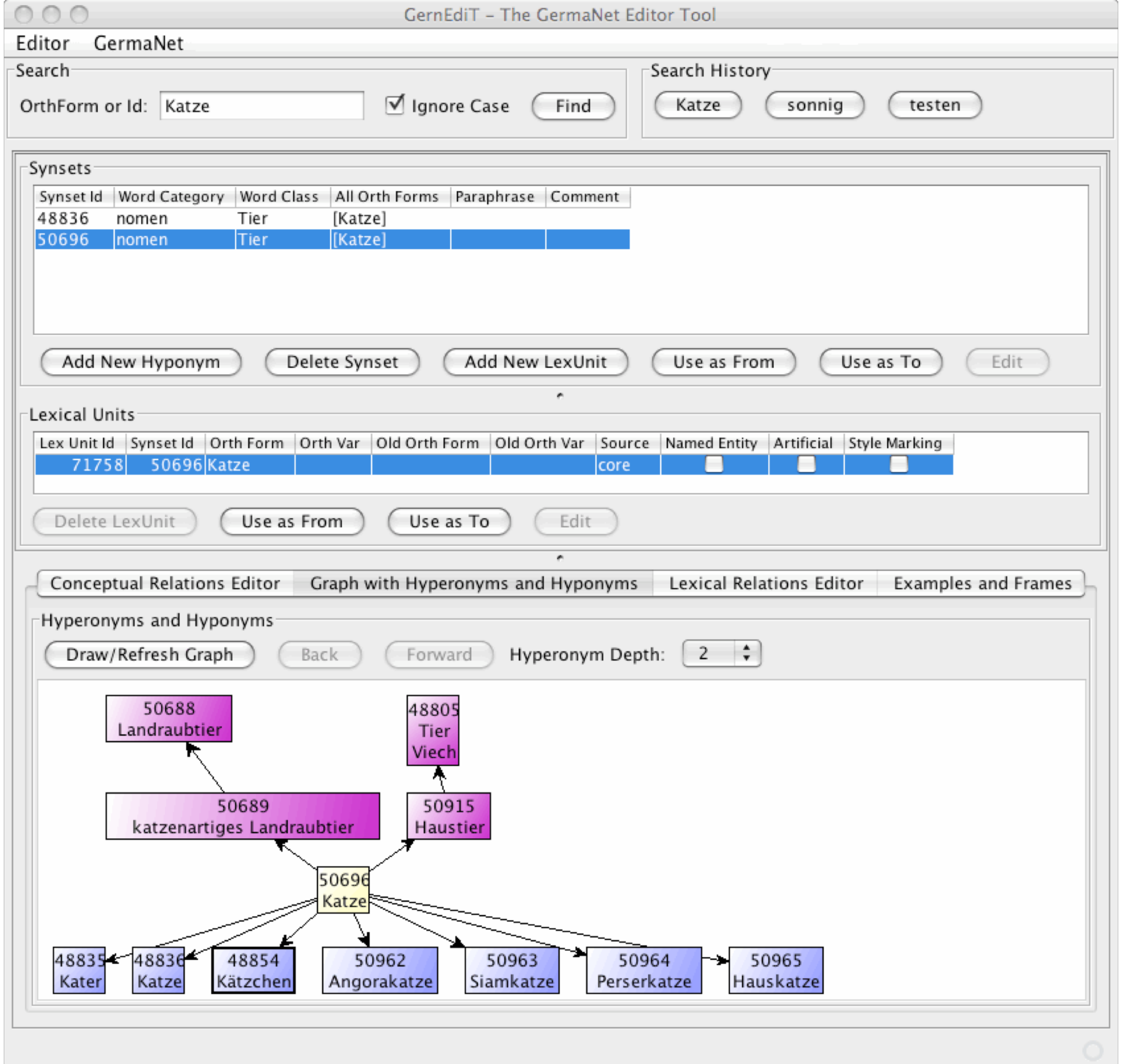
Figure 2 **Co-authorship graph of NiMCS and related research.** Nodes represent authors; edges represent co-authorship. Graph layout uses the ForceAtlas2 algorithm. Clusters are calculated via Louvain modularity and delineated by color. Frequency of co-authorship is calculated via Eigenvector centrality and represented by size.

Image from Alex Garnett, Grace Lee and Judy Illes. 2013. *Publication trends in neuroimaging of minimally conscious states*. PeerJ.

GermaNet Graph - directed graph

From

http://www.sfs.uni-tuebingen.de/lsd/documents/illustrations/GernEdiT-screenshot-large.gif



OrthForm or Id: Katze

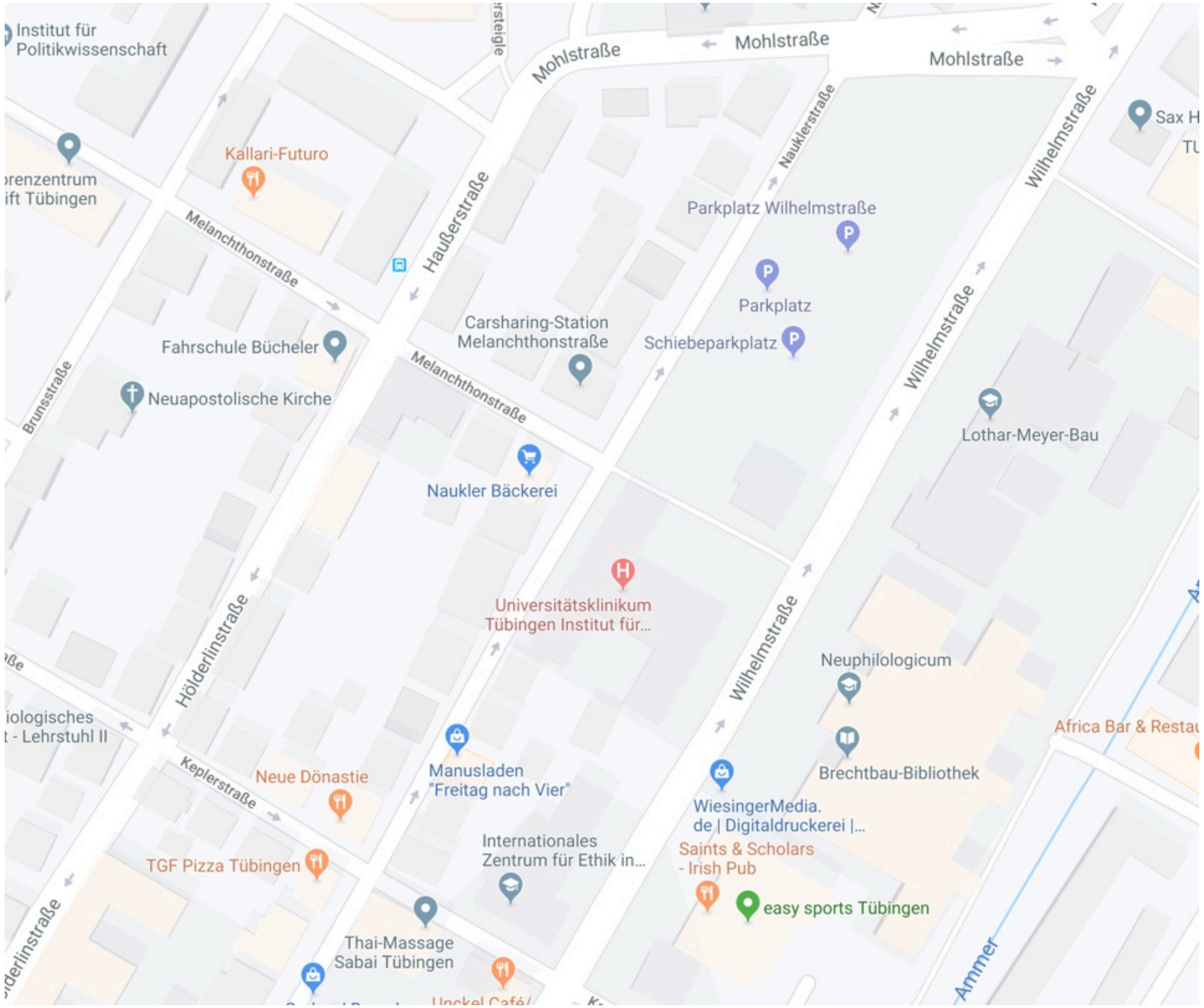
☑ Ignore Case

Synset Id	Word Category	Word Class	All Orth Forms	Paraphrase	Comment
48836	nomen	Tier	[Katze]		
50696	nomen	Tier	[Katze]		

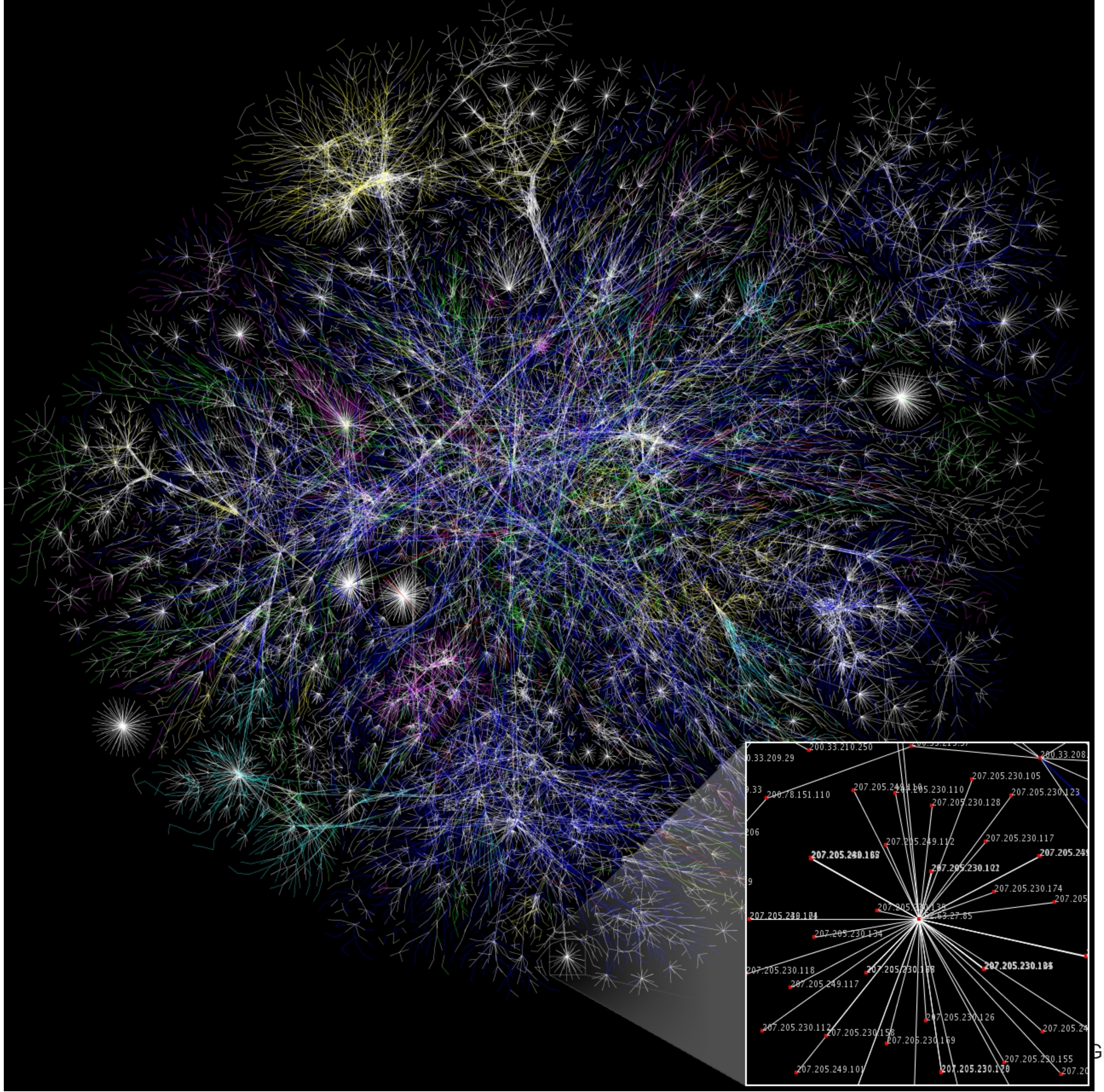
Lex Unit Id	Synset Id	Orth Form	Orth Var	Old Orth Form	Old Orth Var	Source	Named Entity	Artificial	Style Marking
71758	50696	Katze				core	☐	☐	☐

Hyperonym Depth: 2

City Map - mixed graph



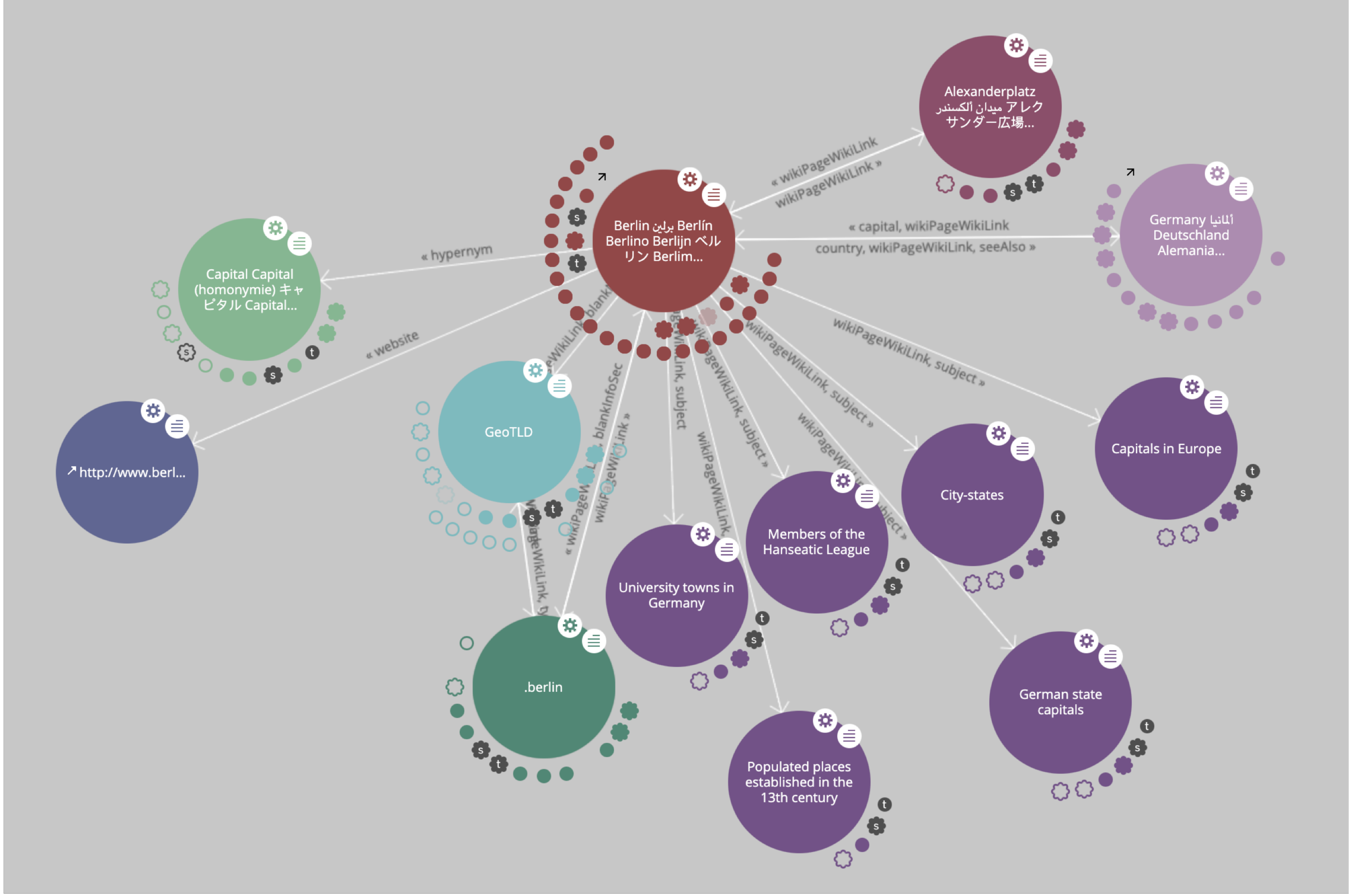
Internet – undirected graph



https://en.wikipedia.org/wiki/Information_visualization#/media/File:Internet_map_1024.jpg

Mixed graph

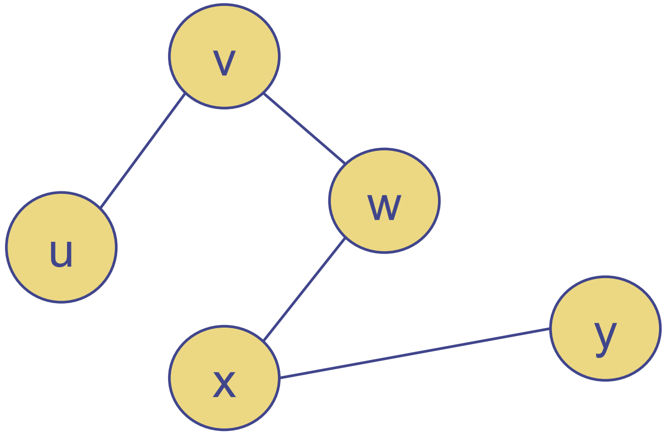
http://dbpedia.org/page/Berlin

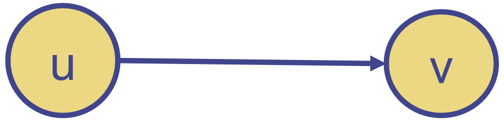


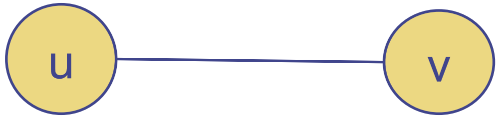
http://en.lodlive.it/?http%3A%2F%2Fdbpedia.org%2Fresource%2FBerlin

Graphs

- A graph G is a set V of vertices – together with a collection E of pairwise connections between vertices from V, called edges
- Graphs are a way of representing relationships that exist between pairs of objects
- Edges in a graph are either directed or undirected
  - An edge (u, v) is directed from u to v if the pair (u, v) is ordered, with u preceding v
  - An edge (u, v) is undirected if the pair (u, v) is not ordered

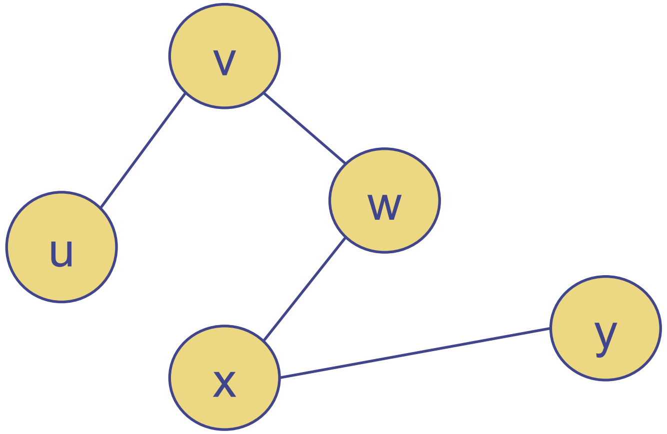




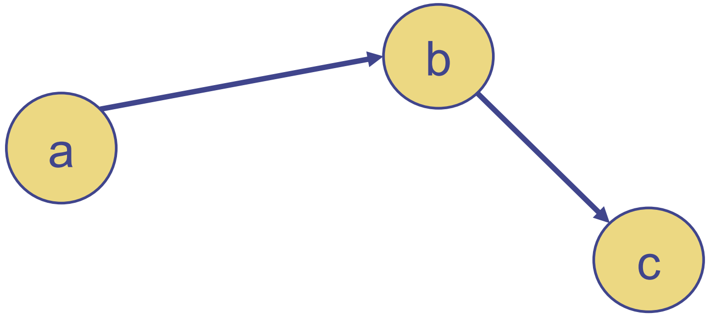


Types of Graphs

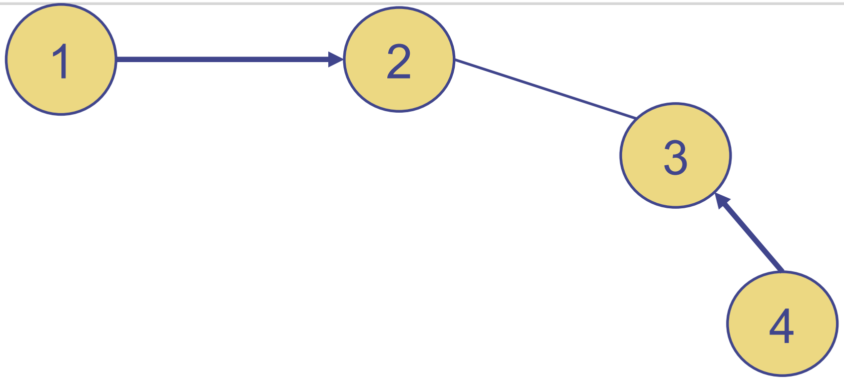
- undirected graph: all the edges in the graph are undirected



- directed graph (digraph): all the edges in the graph are directed

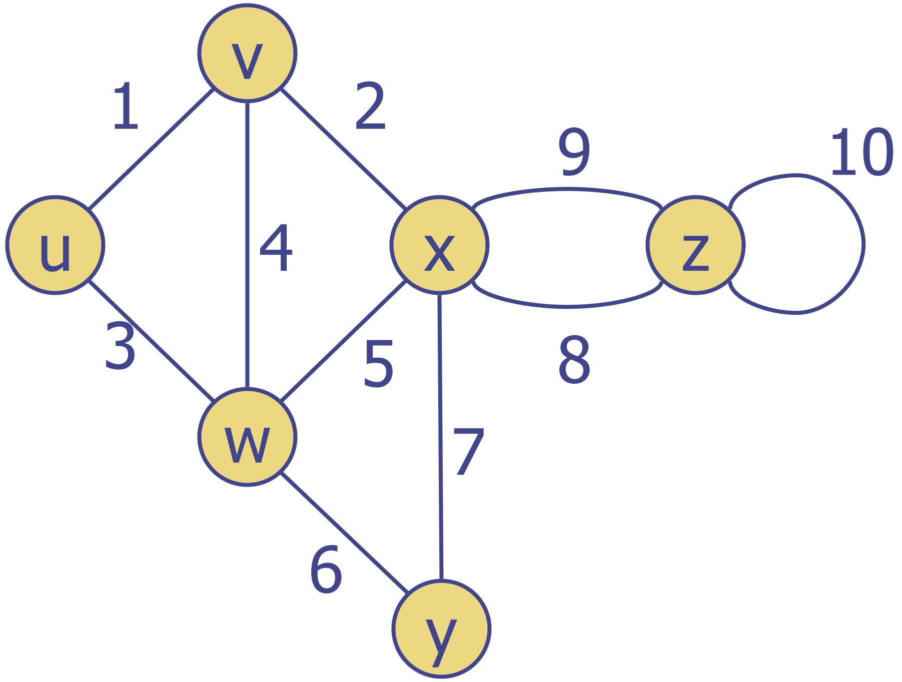


- mixed graph: has both directed and undirected edges



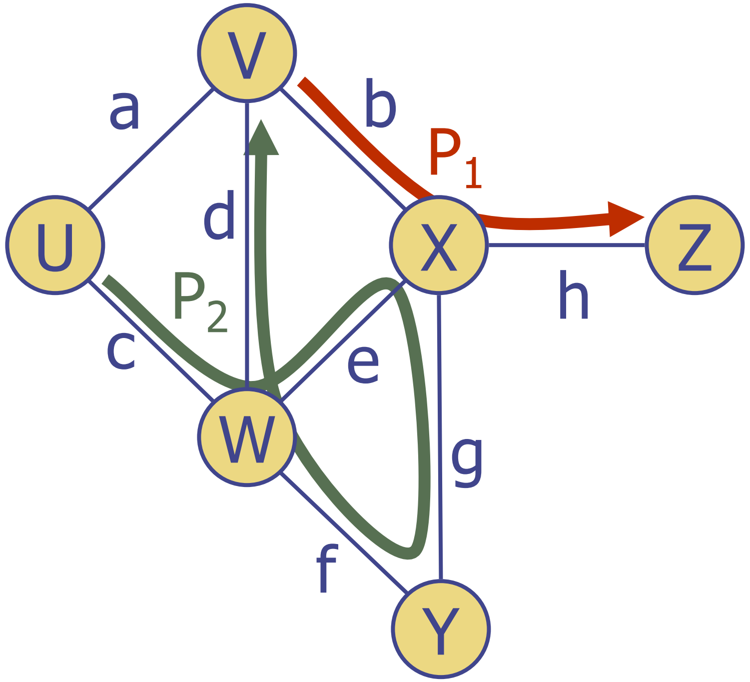
Graph Terminology

- Two vertices joined by an edge are called the end vertices/endpoints of the edge
  - u and v are the endpoints of edge 1
- Two vertices u and v are adjacent if there is an edge whose end vertices are u and v
  - v and x are adjacent
- An edge is called incident to a vertex if the vertex is one of the edge's endpoints
  - edges 1, 2 and 4 are incident to v
- The degree of a vertex, $\deg(v)$, is the number of incident edges of v: v has degree 3
- Edges with the same endpoints are called parallel edges:
  - 8 and 9 are parallel edges
- An edge is a self-loop is its two endpoints coincide:
  - 10 is a self-loop



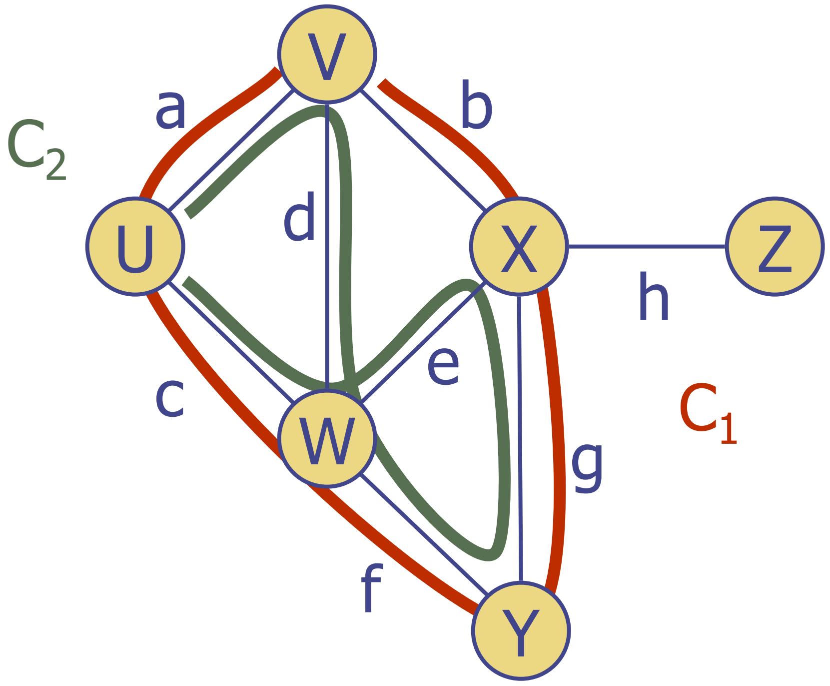
Graph Terminology (cont'd)

- A path is a sequence of alternating edges and vertices that
  - Starts with a vertex
  - Ends with a vertex
  - Each edge is incident to its predecessor and successor vertex
- A path is simple if each vertex in the path is distinct
- Examples of paths
  - $P_1 = (V, b, X, h, Z)$ is a simple path
  - $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ not a simple path because W appears twice



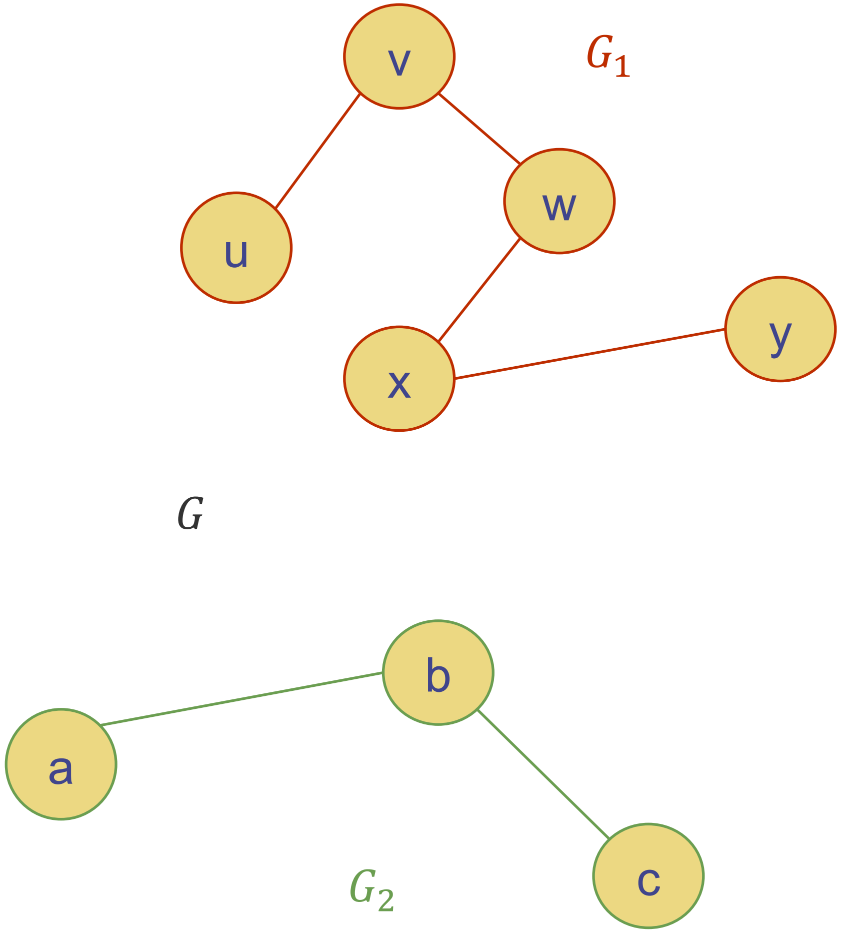
Graph Terminology (cont'd)

- A cycle is a path that
  - Starts and ends at the same vertex
  - Includes at least one edge
- A cycle is simple if all its vertices are distinct, except for the first and the last vertex
- Examples of cycles
  - $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ is a simple cycle
  - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ is not a simple cycle because C_2 goes twice through W



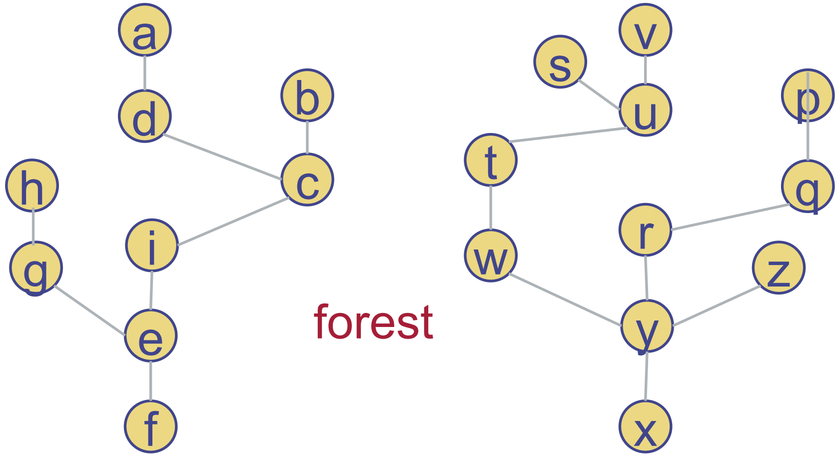
Graph Terminology (cont'd)

- A vertex u reaches a vertex v, and v is reachable from u if there is a path from u to v
  - u reaches y in G_1
  - u does not reach b in G
- A graph is connected if for any two vertices there is a path between them
  - G_1 and G_2 are connected graphs
  - G is not a connected graph
- A subgraph of a graph of G is a graph whose vertices and edges are subsets of the vertices and edges of G
  - G_1 and G_2 are subgraphs of G
- If a graph is not connected, its maximal connected subgraphs are called the connected components of G
  - G_1 and G_2 are the connected components of G

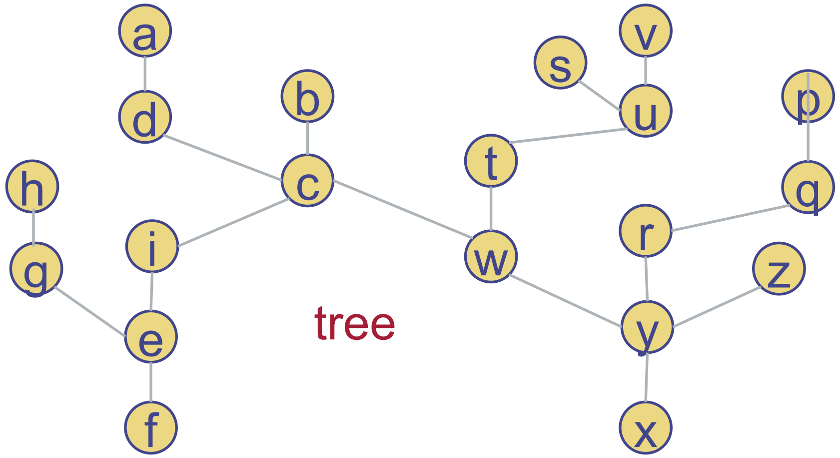


Graph Terminology (cont’d)

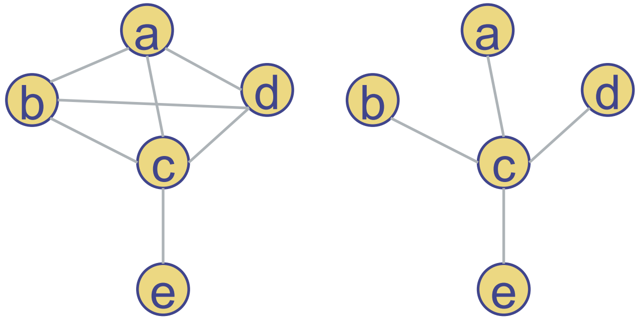
- a spanning subgraph of a graph G is a subgraph of G containing all the vertices of G
- A forest is a disconnected graph without cycles
- A tree is a connected forest – that is – a connected graph without cycles
- A spanning tree of a graph is a spanning subgraph that is a tree



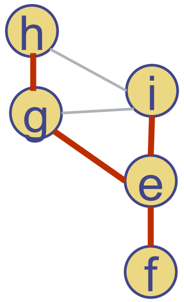
forest



tree



spanning subgraph



spanning tree

Graph Properties

- Property 1. If G is a graph with m edges and vertex set V, then

$$\sum_{v \in V} \deg(v) = 2m$$

- Justification. Any edge (u, v) is counted twice in the summation:
  - Once for its endpoint u
  - Once for its endpoint v
- The total contribution of the edges to the degrees of the vertices is twice the number of edges.

Graph Properties (cont'd)

- Property 2. If G is a simple undirected graph with n vertices and m edges, then

$$m \leq \frac{n(n-1)}{2}$$

- Justification. G is simple, meaning that –
  - there are no edges that have the same endpoints (no parallel edges)
  - there are no self-loops
  - then the maximum degree of a vertex in G is $n - 1$
  - according to property 1, $2m \leq n(n-1) \Rightarrow m \leq \frac{n(n-1)}{2}$

The Graph ADT

The Graph ADT

- A graph is a collection of vertices and edges
- Can be modelled as a combination of three data types: `Vertex`, `Edge` and `Graph`
- class `Vertex`
  - Lightweight object storing the information provided by the user
  - The `element()` method provides a way to retrieve the stored information
- class `Edge`
  - Another lightweight object storing an associated object - the cost
  - The `element()` method provides a way to retrieve the cost of the edge
  - `endpoints()` method: returns a tuple (u, v) where u and v are the Vertex objects
  - `opposite(v)` method: assuming vertex v is one endpoint of an edge, return the other endpoint

The Graph ADT (cont'd)

- class Graph: can be either undirected or directed – flag provided to the constuctor

`vertex_count()`	returns the number of vertices of the graph
`vertices()`	returns an iteration of all the vertices of the graph
`edge_count()`	returns the number of edges of the graph
`edges()`	returns an interation of all the edges of the graph
`get_edge(u,v)`	returns the edge from vertex u to vertex v, if one exists, otherwise None
`degree(v)`	returns the number of edges incident to vertex v
`incident_edges(v)`	returns an iteration of all edges incident to vertex v
`insert_vertex(v, x=None)`	create and return a new `Vertex` storing element x
`insert_edge(u,v, x=None)`	create and return a new `Edge` from vertex u to vertex v, storing x
`remove_vertex(v)`	remove vertex v and all its incident edges from the graph
`remove_edge(e)`	remove edge e from the graph

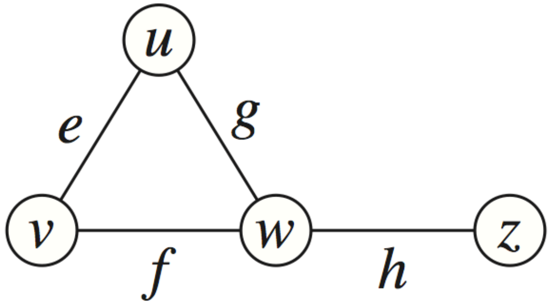
Data Structures for Graphs

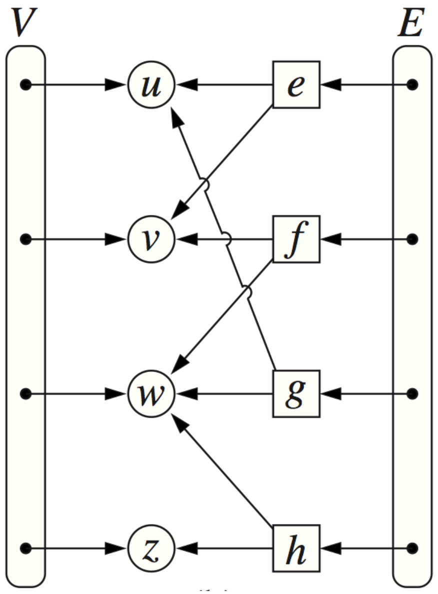
Data Structures for Graphs

- Four data structures for representing a graph
  1. Edge list
  2. Adjacency list
  3. Adjacency map
  4. Adjacency matrix
- In each representation
  - Same: maintain a collection to store the vertices of a graph
  - Different: organize the edges

Edge List Structure

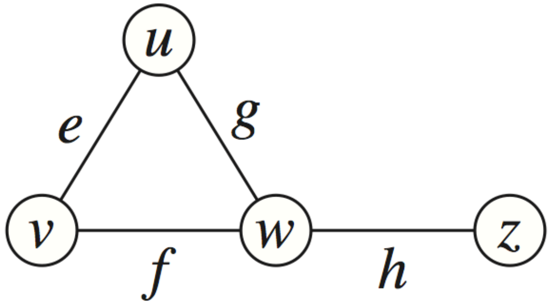
- In an edge list, we maintain
  - an unordered list V to store all vertex objects
  - an unordered list E to store all edge objects
- To support the methods of the Graph ADT, assume:
  - `Vertex`
    - A reference to element x to support the `element()` method
    - A reference to the position of the vertex instance in the list V – for efficient vertex removal
  - `Edge`
    - A reference to element x, to support the `element()` method
    - A reference to the position of the edge instance in list E – for efficient edge removal
    - References to the vertex objects associated with the endpoints of e

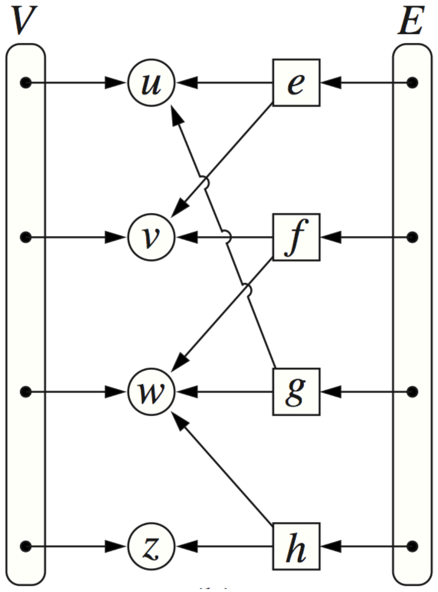




Edge List Structure (cont'd)

- In an edge list, we maintain
  - an unordered list V to store all vertex objects
  - an unordered list E to store all edge objects
- A very simple structure, though not very efficient:
  - locating a particular edge (u, v) - traversing the entire edge list
  - obtaining the set of all edges incident to a vertex v – again, traverse then entire edge list

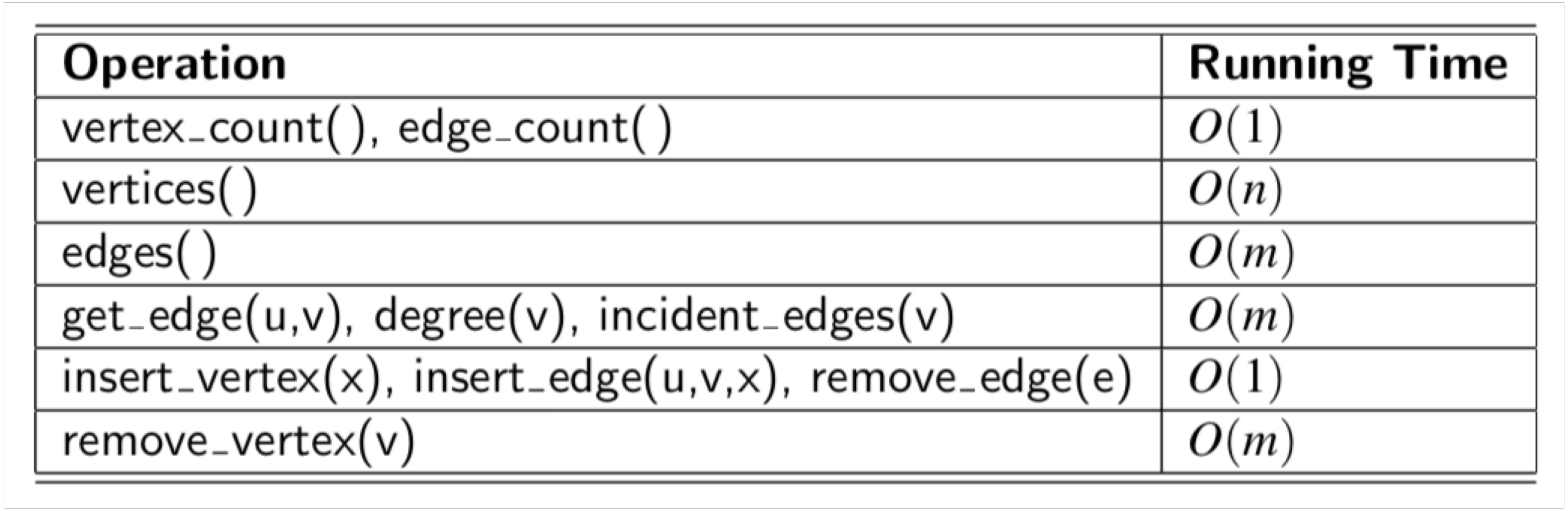




Edge List Structure – Performance

- Space usage
  - $O(n + m)$ for a graph with n vertices and m edges
  - Assuming each individual vertex or edge uses $O(1)$ space
  - The lists V and E use space proportional to their number of entries

Edge List Structure – Performance (cont'd)

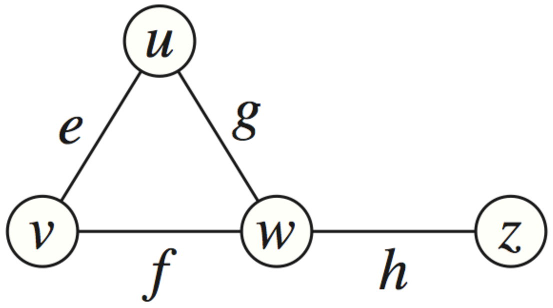


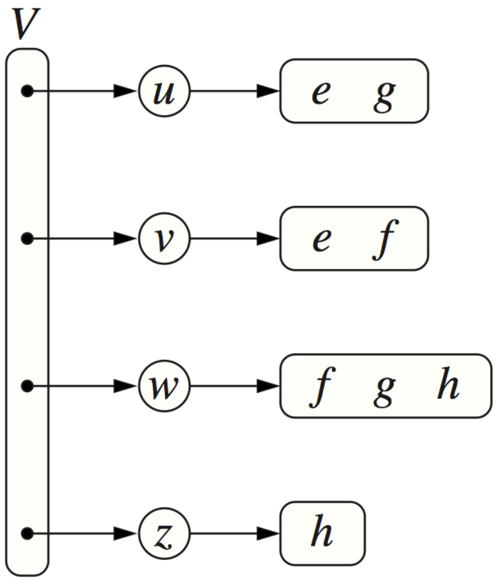
Operation	Running Time
vertex_count( ), edge_count( )	$O(1)$
vertices( )	$O(n)$
edges( )	$O(m)$
get_edge(u,v), degree(v), incident_edges(v)	$O(m)$
insert_vertex(x), insert_edge(u,v,x), remove_edge(e)	$O(1)$
remove_vertex(v)	$O(m)$

- `get_edge(u, v), degree(v), incident_edges(v)` could be implemented more efficiently than $O(m)$
- `remove_vertex(v)` also entails removing all the edges incident to v – otherwise the edges would point to a non-existing vertex of the graph – hence $O(m)$

Adjacency List Structure

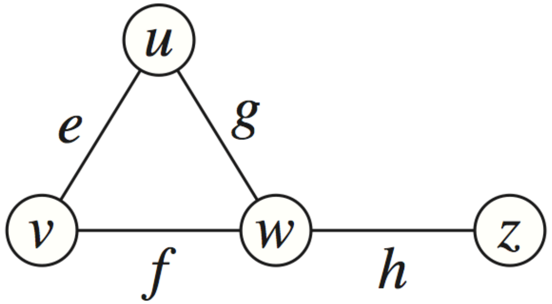
- In an adjacency list, we maintain
  - For each vertex, a separate list containing those edges tha are incident to the vertex
- To support the methods of the Graph ADT, assume:
  - `Vertex`
    - A reference to element x to support the `element()` method
    - A reference to the position of the vertex instance in the list V – for efficient vertex removal
    - A list $I(v)$ – the incidence list of v – containing the edges that are incident to v
  - Edge
    - A reference to element x, to support the `element()` method
    - References to the vertex objects associated with e's endpoints
    - References to the positions of the edge instance in lists $I(u)$ and $I(v)$ – for efficient edge removal

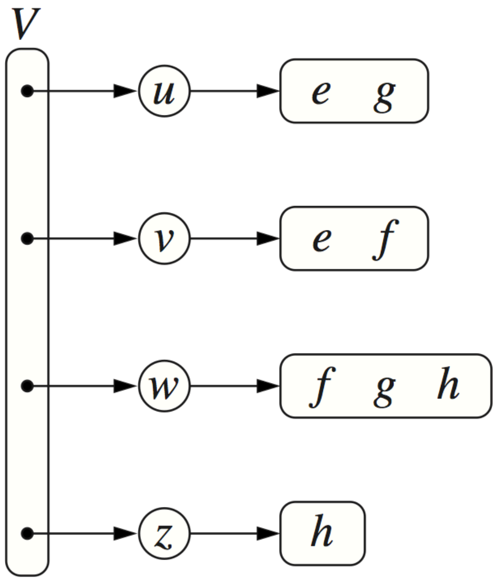




Adjacency List Structure (cont’d)

- In an adjacency list, we maintain
  - For each vertex, a separate list containing those edges tha are incident to the vertex
- Benefits compared to the edge list
  - The $I(v)$ list of each node v contains exactly the edges that should be reported by `incident_edges(v)`
  - Iterate $I(v)$ in $O(\deg(v))$ time instead of iterating the full edge list – the best possible outcome for any graph representation, since there are $\deg(v)$ edges to report

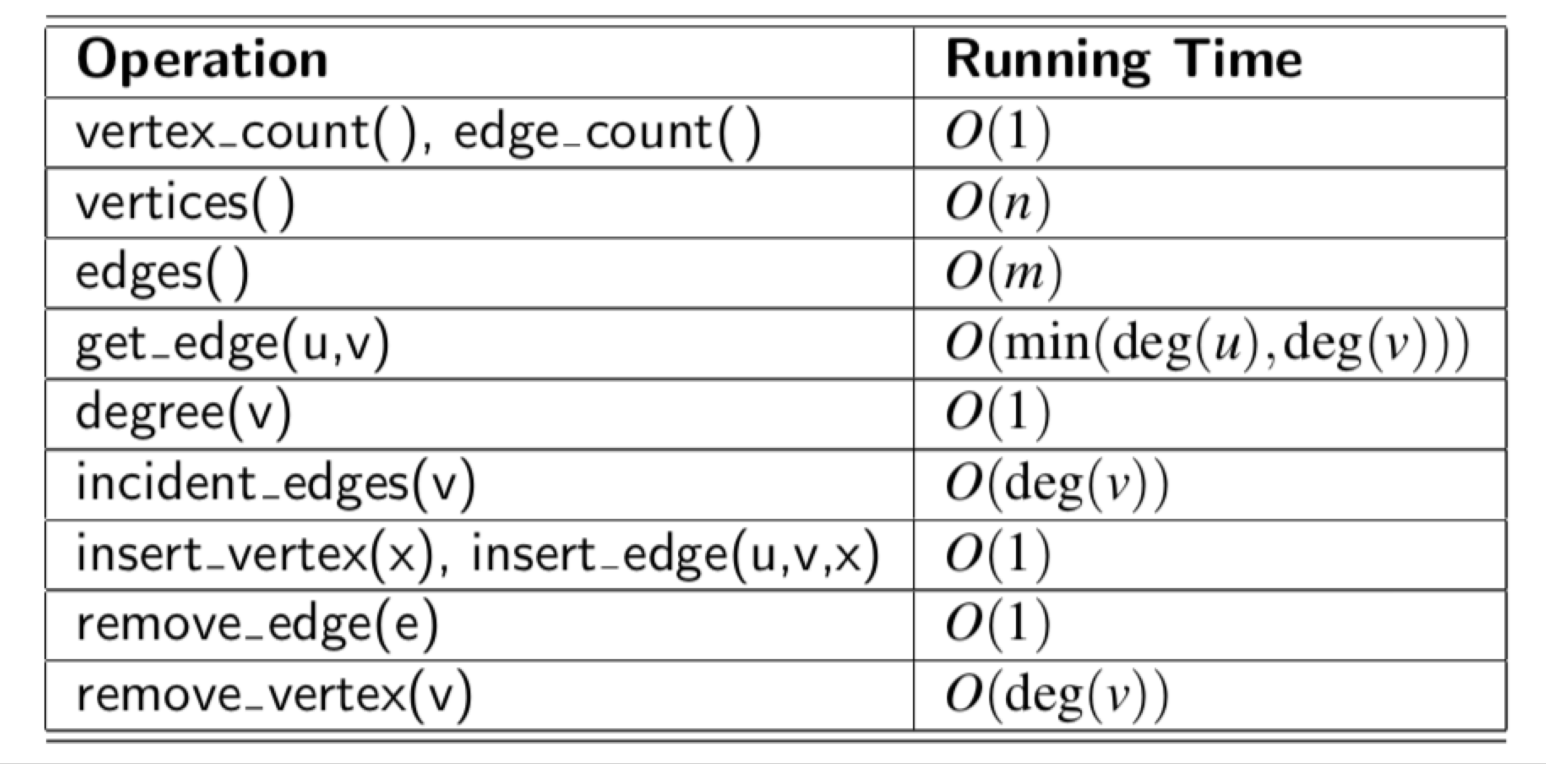




Adjacency List Structure - Performance

- Space usage: asymptotically, the same as the edge list structure
  - $O(n+m)$ for a graph with n vertices and m edges
  - The primary vertex list uses $O(n)$ space
  - The sum of all secondary lists containing the edges incident to each vertex is $O(m)$
    - An undirected edge (u, v) is referenced both in $I(u)$ and in $I(v)$, but its presence in the graph results only in a constant amount of additional space

Adjacency List Structure – Performance (cont'd)

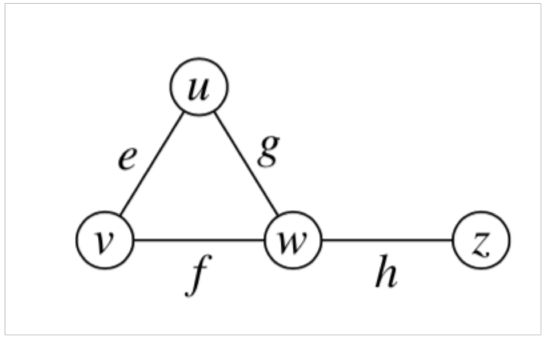


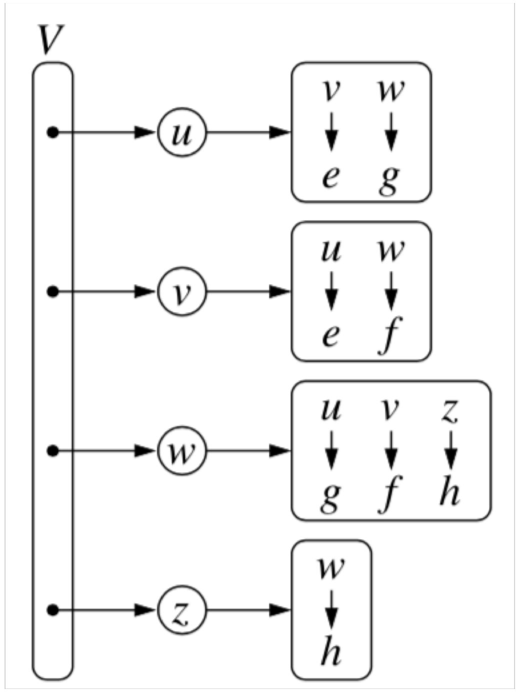
Operation	Running Time
vertex_count(), edge_count()	$O(1)$
vertices()	$O(n)$
edges()	$O(m)$
get_edge(u,v)	$O(\min(\deg(u), \deg(v)))$
degree(v)	$O(1)$
incident_edges(v)	$O(\deg(v))$
insert_vertex(x), insert_edge(u,v,x)	$O(1)$
remove_edge(e)	$O(1)$
remove_vertex(v)	$O(\deg(v))$

- `get_edge(u,v)` – we can look for the edge in either the list of u or that of v – take the shortest
- Because we are storing the positions of e in $I(u)$ and $I(v)$, removing an edge takes $O(1)$ time
- To remove a vertex v we need to also remove all its incident edges – but there are all in $I(v)$, so `remove_vertex(v)` runs in $O(\deg(v))$ time

Adjacency Map Structure

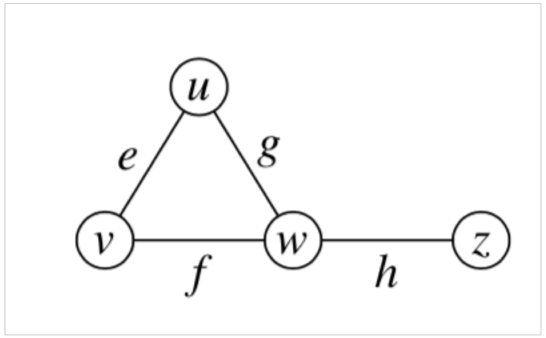
- In an adjacency map, we maintain
  - For each vertex v, a separate hash-map
  - Each entry has as key the vertex that is opposite to v, and as value the edge which has u and v as endpoints
- To support the methods of the Graph ADT, assume:
  - `Vertex`
    - A reference to element x to support the `element()` method
    - A reference to the position of the vertex instance in the list V – for efficient vertex removal
    - A hashmap $I(v)$ – containing (vertex, edge) pairs where the vertices are the opposites of v and the edges are the edges incident to v
  - `Edge`
    - A reference to element x, to support the `element()` method
    - References to the vertex objects associated with e's endpoints

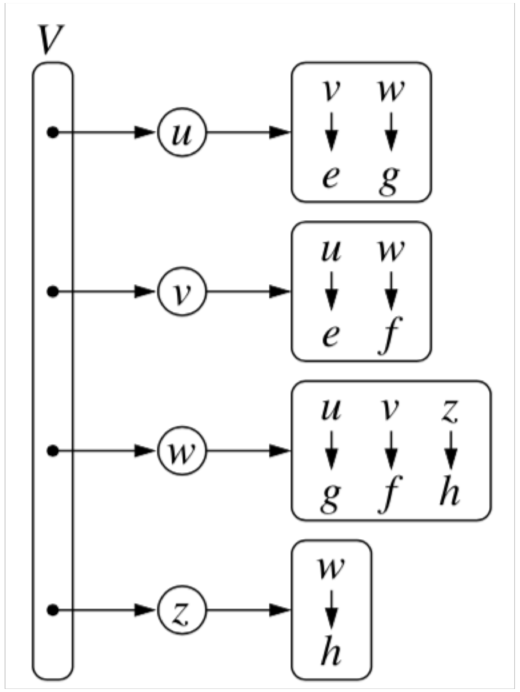




Adjacency Map Structure

- In an adjacency map, we maintain
  - For each vertex v, a separate hash-map
  - Each entry has as key the vertex that is opposite to v, and as value the edge which has u and v as endpoints
- Benefits compared to the adjacency list
  - `get_edge(u,v)` can be implemented in expected $O(1)$ time by searching for vertex u as a key in $I(v)$ or vice-versa
  - this is better than in the adjacency list case, where the best case performance was $O(\min(\deg(u), \deg(\mathrm{v})))$

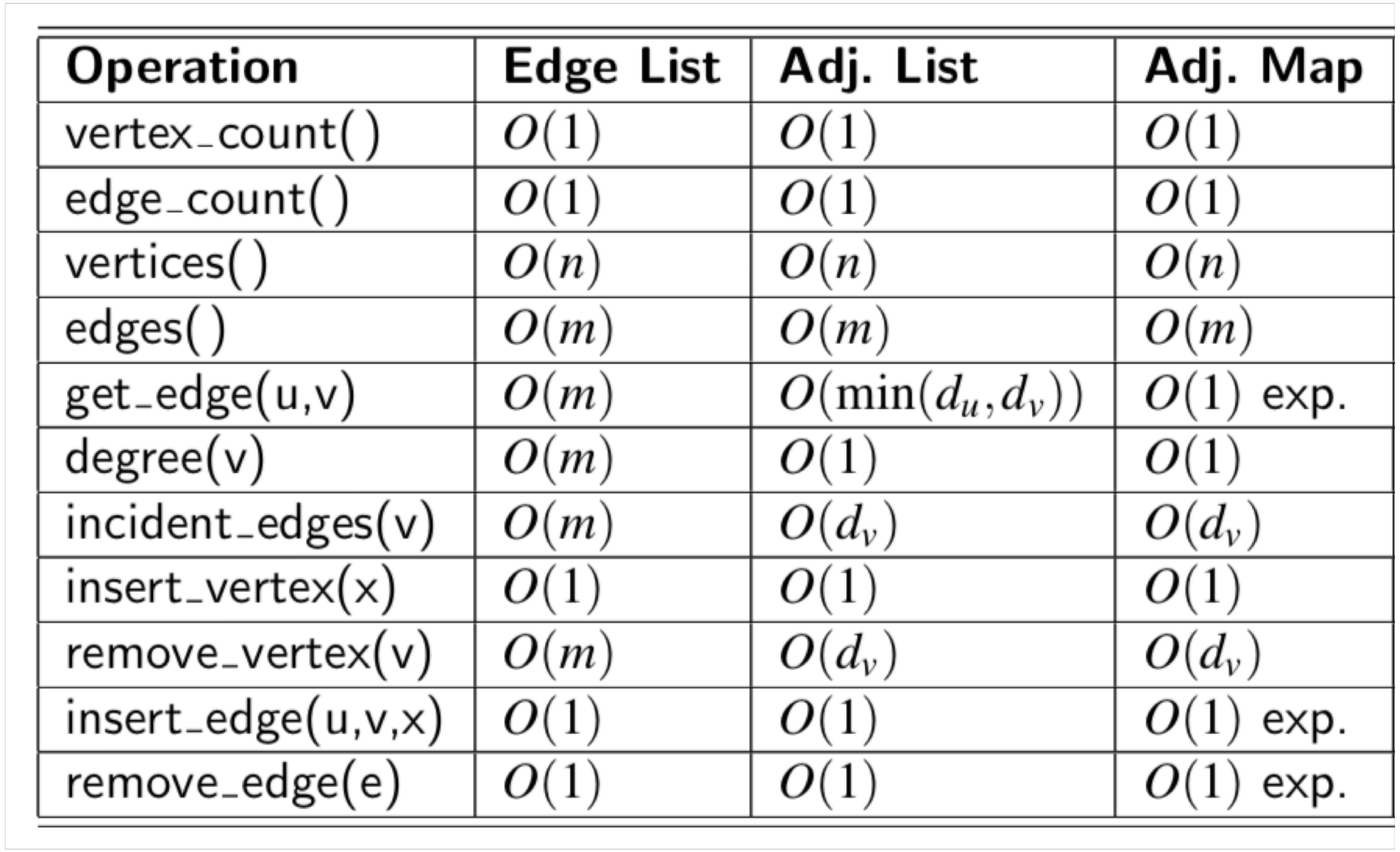




Adjacency Map Structure - Performance

- Space usage
    - $O(n+m)$, just like the adjacency list
    - For each vertex $\mathrm{u}, I(u)$ - an adjacency map uses $O(\deg(u))$ space

Adjacency Map Structure – Performance (cont’d)

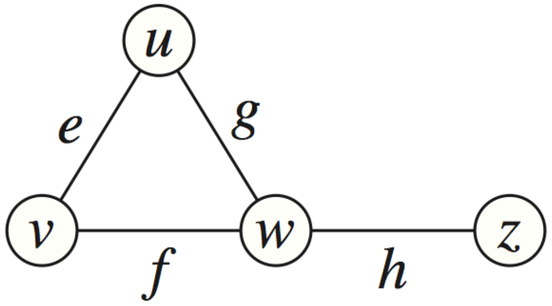


Operation	Edge List	Adj. List	Adj. Map
vertex_count()	$O(1)$	$O(1)$	$O(1)$
edge_count()	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.
degree(v)	$O(m)$	$O(1)$	$O(1)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.

- d_v - the degree of v
- an adjacency map achieves essentially optimal running times for all methods, making in an excellent all-purpose choice as a graph representation structure

Adjacency Matrix Structure

- In an adjacency matrix structure, we maintain
  - An $n \times n$ matrix A of edges, storing references to edges
  - $A[i, j]$ stores a reference to the edge (u, v) if it exists, where u is the vertex with index i and v is the vertex with index j
    - if there is no such edge, then A[i,j] = None
  - A is symmetric if the graph is undirected
  - An edge between a given pair of vertices can be retrieved in worst-case constant time

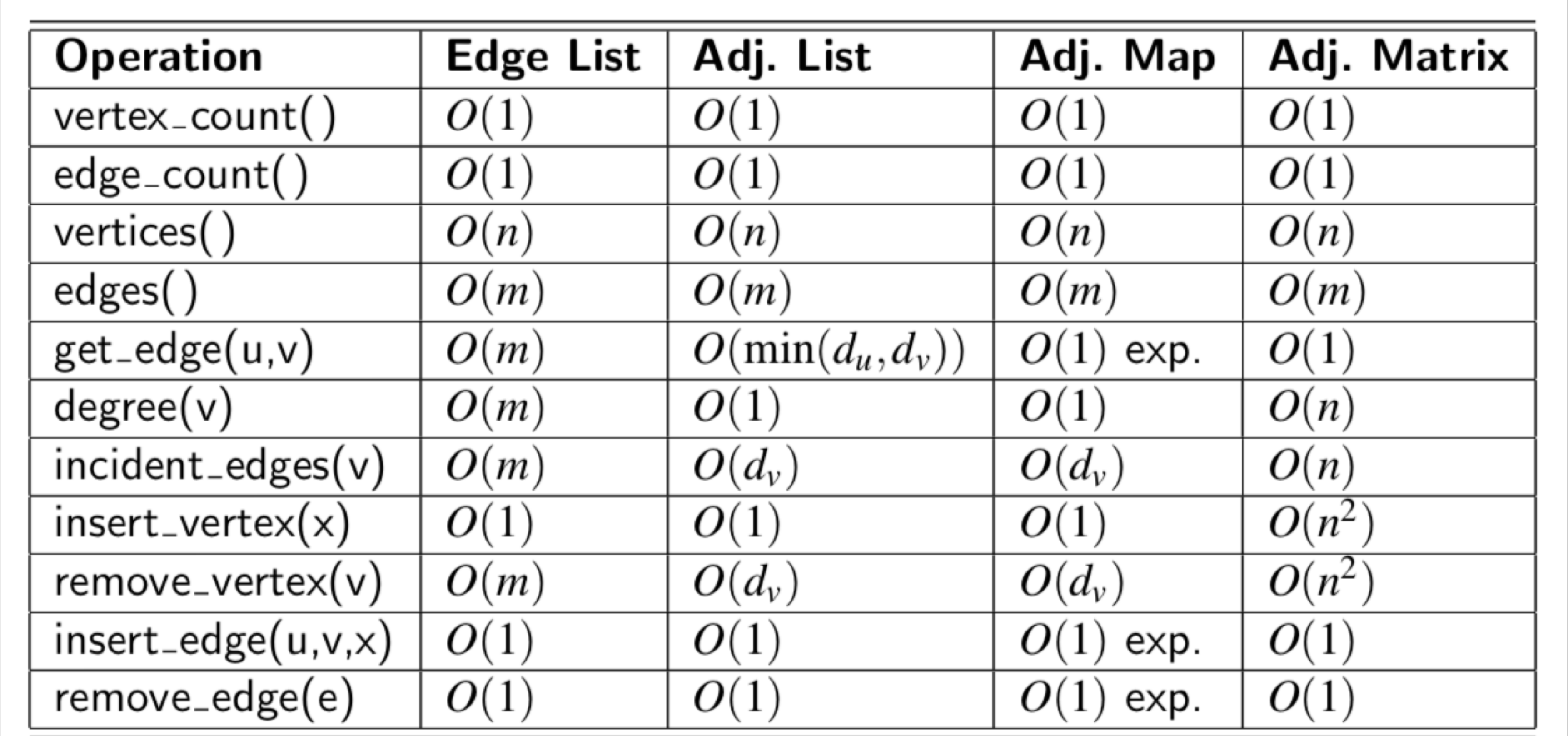


$$\begin{array}{ccc|c|c|c|c|}
 & & & 0 & 1 & 2 & 3 \\
\hline
u & \longrightarrow & 0 & & e & g & \\
\hline
v & \longrightarrow & 1 & e & & f & \\
\hline
w & \longrightarrow & 2 & g & f & & h \\
\hline
z & \longrightarrow & 3 & & & h & \\
\hline
\end{array}$$

Adjacency Matrix Structure - Performance

- Space usage
  - $O(n^2)$ space, much worse than the $O(n + m)$ needed for the other three structures
  - Although if the graph is dense the number of edges is proportional to $O(n^2)$
  - In practice, most real-word graphs are sparse – making the adjacency matrix structure inefficient, since it will store many None values
  - If a graph is dense, a adjacency matrix might be more efficient then an adjacency list or map
  - Particularly if edges have no auxiliary data, then an adjacency matrix can be implemented using a Boolean matrix, using 1 bit to store information about each edge slot, e.g. $A[i, j] = True$ if and only if (u, v) is an edge in the graph

Adjacency Matrix Structure – Performance (cont’d)



Operation	Edge List	Adj. List	Adj. Map	Adj. Matrix
vertex_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
edge_count()	$O(1)$	$O(1)$	$O(1)$	$O(1)$
vertices()	$O(n)$	$O(n)$	$O(n)$	$O(n)$
edges()	$O(m)$	$O(m)$	$O(m)$	$O(m)$
get_edge(u,v)	$O(m)$	$O(\min(d_u, d_v))$	$O(1)$ exp.	$O(1)$
degree(v)	$O(m)$	$O(1)$	$O(1)$	$O(n)$
incident_edges(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n)$
insert_vertex(x)	$O(1)$	$O(1)$	$O(1)$	$O(n^2)$
remove_vertex(v)	$O(m)$	$O(d_v)$	$O(d_v)$	$O(n^2)$
insert_edge(u,v,x)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$
remove_edge(e)	$O(1)$	$O(1)$	$O(1)$ exp.	$O(1)$

- `get_edge(u,v)` is an $O(1)$ operation
- Several operations are less efficient:
  - `degree(v)`, `incident_edges(v)` – we need to examine all n entries in the row associated with vertex v – $O(n)$
  - `insert_vertex(v)`, `remove_vertex(v)` – the matrix has to be resized - $O(n^2)$

Python Implementation – using an Adjacency Map variant

- Use a Python dictionary to represent each secondary incidence map, $I(v)$
- Use a top-level dictionary D to map each vertex v to its incidence map, $I(v)$
- All the vertices of the graph can be obtained by iterating over the keys of D
- This frees us from having to keep indices for the position of the vertices in the Vertex
- Also, rather than maintaining a separate list of edges, the edges can be found in $O(n + m)$ time by taking the union of the edges found in all the incidence maps

Vertex class

```
#------------------------- nested Vertex class -------------------------
class Vertex:
  """Lightweight vertex structure for a graph."""
  __slots__ = '_element'

  def __init__(self, x):
    """Do not call constructor directly. Use Graph's insert_vertex(x)."""
    self._element = x
  @property
  def element(self):
    """Return element associated with this vertex."""
    return self._element

  def __hash__(self):            # will allow vertex to be a map/set key
    return hash(id(self))
```

__slots__

- By default Python represents each namespace with an instance dictionary of the built-in dict class- this maps identifying names in the scope to the associated objects
- While a dictionary structure supports relatively efficient name lookups, it requires additional memory beyond the raw data that it stores.
- Python provides a more direct mechanism for representing instance namespaces, that avoids the use of an auxiliary dictionary.
- To streamline the representation for all instances of a class, the class should define a class-level member named __slots__ that is assigned a fixed sequence of strings that serve as names for instance variables
- Advisable in particular in any nested classes that are expected to have many instances

__init__

- Whenever an instance of the Vertex class is created using a statement of the type v = Vertex("A"), a special method called __init__ is called
- __init__ serves as the constructor of the class
- It is responsible primarily for establishing the state of the new object – e.g. set up the _element instance variable in the case of Vertex, set up the _origin, _destination and _element in the case of Edge
- By convention a single leading underscore in the name of a data member, such as _element implies that it is intended as nonpublic; users of a class should not directly access such members

@property

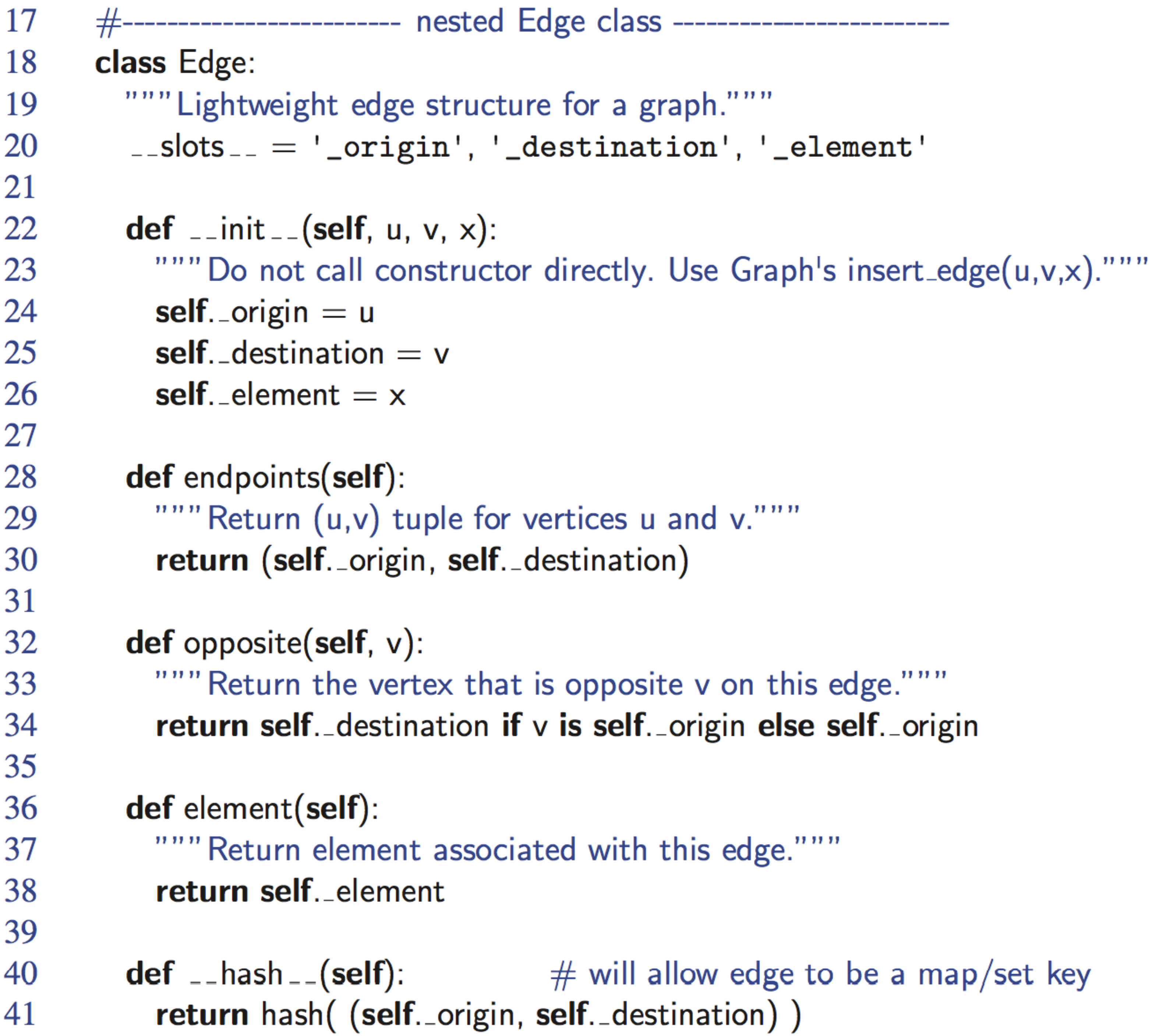
- @property is a decorator which indicates that the element(self) method is a “getter” method, and that the name of the property is the method name only – e.g. only element
- A decorator is a function which receives another function as an argument
- The behavior of the argument function is extended by the decorator without actually modifying it
- The element of a vertex can then be obtained using x.element
- There is also a corresponding way of creating a setter using the @f.setter decorator

```
@element.setter
def element(self, el):
    self._element = el
```

__hash__

- Standard Python mechanism for computing hash codes – hash(x) returns an integer value that serves as a hash code for object x
- Only immutable data types are hashable in Python – to ensure that the object’s hash code remains constant during the lifetime of the object
  - It an object is inserted into a hash table, and then its hash code would change, then a different object would be retrieved from the hash table
- Instances of user-defined classes are unhashable by default
- A function that computes the hash code can be implemented via the __hash__ method within the class
  - The returned hash code should reflect the immutable attributes of an instance (e.g. _element would not make for a good attribute for hashing, it might be updated)
- Also, if x == y, then hash(x) == hash(y)

Edge Class



```
#------------------------ nested Edge class ------------------------
class Edge:
  """Lightweight edge structure for a graph."""
  __slots__ = '_origin', '_destination', '_element'

  def __init__(self, u, v, x):
    """Do not call constructor directly. Use Graph's insert_edge(u,v,x)."""
    self._origin = u
    self._destination = v
    self._element = x

  def endpoints(self):
    """Return (u,v) tuple for vertices u and v."""
    return (self._origin, self._destination)

  def opposite(self, v):
    """Return the vertex that is opposite v on this edge."""
    return self._destination if v is self._origin else self._origin

  def element(self):
    """Return element associated with this edge."""
    return self._element

  def __hash__(self):            # will allow edge to be a map/set key
    return hash( (self._origin, self._destination) )
```

self

- self identifies the instance upon which a method is invoked
- self is also used to store the instance variables that reflect its current state
- self._element refers to an instance variable named _element that is stored as part of that particular Vertex's state
- There is a difference between a method signature as declared within a class vs. that used by a caller:
  - E.g. from the user's perspective the opposite() method takes one parameter, the Vertex v, while endpoints() takes no parameters
  - However, within the class definition self in an explicit parameter, making opposite() have two parameters, and endpoints() one parameter
- The Python interpreter will automatically bind the instance upon which the method is invoked to the self parameter

Graph Class, part 1

```
class Graph:
  """Representation of a simple graph using an adjacency map."""

  def __init__(self, directed=False):
    """Create an empty graph (undirected, by default).

    Graph is directed if optional paramter is set to True.
    """
    self._outgoing = { }
    # only create second map for directed graph; use alias for undirected
    self._incoming = { } if directed else self._outgoing

  def is_directed(self):
    """Return True if this is a directed graph; False if undirected.

    Property is based on the original declaration of the graph, not its contents.
    """
    return self._incoming is not self._outgoing # directed if maps are distinct

  def vertex_count(self):
    """Return the number of vertices in the graph."""
    return len(self._outgoing)

  def vertices(self):
    """Return an iteration of all vertices of the graph."""
    return self._outgoing.keys( )

  def edge_count(self):
    """Return the number of edges in the graph."""
    total = sum(len(self._outgoing[v]) for v in self._outgoing)
    # for undirected graphs, make sure not to double-count edges
    return total if self.is_directed( ) else total // 2

  def edges(self):
    """Return a set of all edges of the graph."""
    result = set( )          # avoid double-reporting edges of undirected graph
    for secondary_map in self._outgoing.values( ):
      result.update(secondary_map.values( ))       # add edges to resulting set
    return result
```

Python Generators

- The most convenient technique for creating iterators in Python is through the use of generators
- A generator is implemented with a syntax that is very similar to a function, but instead of returning values, a yield statement is executed to indicate each element of a sequence
- It is illegal to combine return and yield statements in the same implementation
- Lazy evaluation: the results are only computed if requested, the entire sequence need not reside in memory at one time – generators can produce infinite sequences of values
- Generator comprehensions do not create temporary lists

Graph Class, part 2

```
    def get_edge(self, u, v):
        """Return the edge from u to v, or None if not adjacent."""
        return self._outgoing[u].get(v)            # returns None if v not adjacent

    def degree(self, v, outgoing=True):
        """Return number of (outgoing) edges incident to vertex v in the graph.

        If graph is directed, optional parameter used to count incoming edges.
        """
        adj = self._outgoing if outgoing else self._incoming
        return len(adj[v])

    def incident_edges(self, v, outgoing=True):
        """Return all (outgoing) edges incident to vertex v in the graph.

        If graph is directed, optional parameter used to request incoming edges.
        """
        adj = self._outgoing if outgoing else self._incoming
        for edge in adj[v].values( ):
            yield edge

    def insert_vertex(self, x=None):
        """Insert and return a new Vertex with element x."""
        v = self.Vertex(x)
        self._outgoing[v] = { }
        if self.is_directed( ):
            self._incoming[v] = { }          # need distinct map for incoming edges
        return v

    def insert_edge(self, u, v, x=None):
        """Insert and return a new Edge from u to v with auxiliary element x."""
        e = self.Edge(u, v, x)
        self._outgoing[u][v] = e
        self._incoming[v][u] = e
```

Thank you.